Embedded Systems Software: Linux

Edited by Colleen Purtell-Tappan Technical Editor

Rethinking Embedded Development with Linux

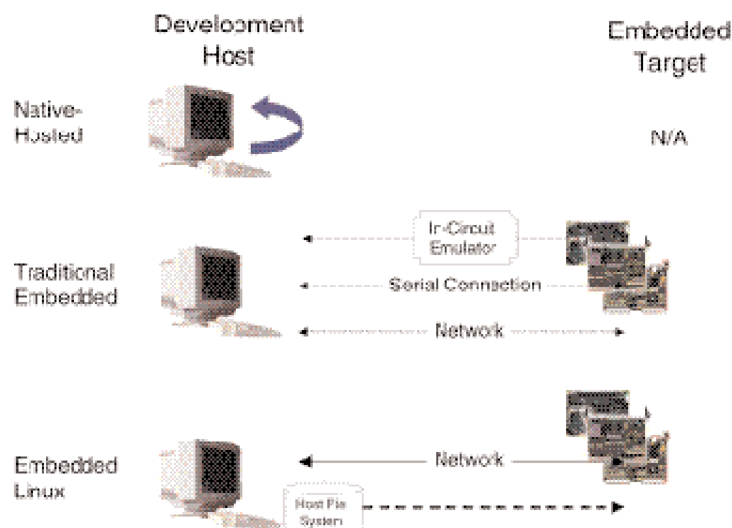
by Bill Weinberg, Technologist, MontaVista Software Inc.

When embarking on an embedded Linux design project for the first time, "old school" developers need to readjust their expectations of how development proceeds and what tools they use. Key areas for re-examination include allocation of host and target capabilities, cross loading, device programming, debugging and target resource use.

Host and Target — A Marriage of Unequals?

The conventional image of computer programming is of a lone engineer hacking away on a PC, coding and testing on the same machine, for eventual deployment on a similar computer. For embedded development, two machine are necessarily involved: the development host, a PC-compatible or UNIX workstation; and the embedded target, actual application hardware or a prototype thereof.

The host traditionally outweighes the target machine in resources and compute power. The host has always been a "real computer", while the target was to varying degrees a "toaster."



Today's breed of smarter, faster, embedded devices eliminate the silicon gap between host and target: 32-bit and 64-bit processors and ASIC cores rival and sometimes outstrip hosted CPUs. By embedding Linux, the once-humble target graduates from toaster status to full computing peer. The networking, file system(s), shell(s), tools and utilities that run on embedded Linux are exactly those found on development hosts, and developers can leverage them as needed, blurring the boundaries between host and target.

Target Connection

While traditional cross development depended on serial, parallel, ICE, ROMulation and sometimes network connections between host and target, most embedded Linux cross development leverages built-in TCP/IP and Ethernet for connectivity. Since most of the CPUs chosen for embedded Linux deployment boast built-in Ethernet, this 10 Mbit or 100 Mbit connection is cheaper, faster and more reliable than any of the old cups-and-string approaches.

Device Programming

In classic embedded development, building an executable proceeded as follows:

- Source files compile to relocatables;
- Object files, libraries, and the kernel itself link into one massive load image;
- A load image gets programmed into EPROM or cross loaded into target RAM, often first needing conversion to a device programmer file format, like Intel HEX or Motorola S-Records;
- Repeat (laboriously) as needed.

Linux, with its intelligent loader, dynamic/shared libraries, and ever-present file system, changes this paradigm as follows:

- Source files compile to relocatable object files in shared directories;
- Object files, static libraries, and dynamic/shared library stubs link intoa recognizable executable (just like on the development host);
- The program loads and executes on the embedded Linux target, resolving dynamic library references either at load time or at run-time, implicitly or explicitly, as needed. By networked-mounting host file systems on the target, the once-explicit cross loading can appear instantaneous.

Now, this processes presupposes the existence of a working Linux target. How did Linux get there in the first place?

- Start with a standard or custom-built bootable Linux kernel file for the target CPU (presumed different from host architecture);
- The host uses the Linux "loop-back" capability to mount a single file as though it were a complete file system, no special tools needed;
- Copy the kernel and any other needed files (from /bin, your applications, etc.) to the mounted file/file system. Don't forget the shared libraries;
- Unmount the loop-back file and get a file suitable for binary device programming or conversion to HEX or S-Records; and
- Use a comparable procedure for building an initial RAM file system image.

File System

Classic embedded designs were ROM-based and diskless. For developers familiar with these designs, the term "file system" conjures up images of "fat and slow," inappropriate for embedding. Linux obligatorily includes a file system, for program loading/resolving paths but not necessarily with rotating media (disk, etc.). Embedded Linux systems most often sport a minimal RAM file system and a ROM or flash file system for boot and storage, at minimal resource cost.

Wither Hardware Debug?

Hardware debugging was once the mainstay of embedded development. Engineering teams shelled out big bucks to put in-circuit emulators (ICE) on developers' desks, and although ICE units have never been very reliable beasts, for many years they were an indispensible accessory.

Today, developers coming to embedded Linux bring expectations of using their old hardware debug tools, and are often frustrated to find that Penguins and ICE don't easily mix.

Why? Consider how ICE-based debug operations work. To set a breakpoint, a hardware debugger sets up a binary comparitor on the CPU address Bus1 or leverages CPU-based breakpoint registers to work for them.

Linux is a virtually-addressed operating system, meaning that user program and the operating system execute in a logical view of system memory. Each address access must be translated from program-centric logical memory to a real physical system address by on-chip Memory Management Units (MMUs). Benefits of virtual addressing include better memory utilization, fine-grained memory protection and R/W control, application-generic memory maps and options for extending RAM to disk with virtual memory.

For simple control loop programs on bare silicon, or with "flat address" embedded executives (like pSOS, VRTX, or VxWorks), ICE works fine. Physical and logical addresses are the same, so a breakpoint set on address 0x1000B010 will always map to that address to trigger on a program fetch or data access. A virtual addressing scheme, however, maps logical program and data memory onto available physical memory in 4 Kbyte pages, with mappings unique to each running process (user program) and to the Linux kernel itself. Thus, logical address 0x1000B010 could be offset anywhere in memory by multiples of 4 Kbytes. When the hardware-oriented emulator is told by a debugger to break on a program address, it has no way of finding the actual target address and will simply set a breakpoint on 0x1000B010 even if the actual physical address is 0xA00D010 (0x1000B010 + 0x90002000).

A few hardware debugging vendors, like Abatron and Applied Microsystems, have solved this problem with software-assist, either by linking target code that requests translations or by adding a special kernel module that "walks" the OS memory translation tables.

The Kernel as a Debugger

Embedded teams can spend upward of $25,000 per developer for tool suites that purport to address key aspects of the development cycle. In reality, many of those tools end up as shelfware, and those that apply well to flat-model kernels fall flat with Linux because they cannot handle virtual addressing.

The open source world that gave birth to Linux proffers myriad tools for hosted and embedded development, but the Linux kernel itself can often be the best tool around. By leveraging the protected POSIX.1 process model and running core files through the GDB debugger, you can —

- Catch illegal writes to program code and protected data segments;
- Prevent overwrite of the kernel and other processes' code;
- Detect unitialized and other stray pointers;
- Protect thread stacks, trap on underflow, and even map in new stack depth;
- Isolate failures to single processes instead of the entire application machine;

Conclusion

Embedded developers are a flexible, forward-looking bunch, and despite the need to reorient themselves technically, they are flocking to Linux like penguins to their nesting ground. They are choosing Linux for the technical advantages cited above, for its greater reliability, for the comprehensive set of standard APIs and to lower their cost of goods sold, and bring their products to market faster.

Bill Weinberg is Director of Marketing for MontaVista Software Inc, 1237 East Arques Ave., Sunnyvale, CA 94085; 408-328-9200; Fax: 408-328-3875; sales@mvista.com; www.mvista.com.

EDITORIAL EVALUATION

Write in Number or Reply Online

I found this article:

Very Useful	Useful	Not Useful
xx	xxx	xxx